Many Sequential Iterative Algorithms Can Be Parallel and (Nearly) Work-efficient

Zheqi Shen
UC Riverside
zheqi.shen@email.ucr.edu

Zijin Wan
UC Riverside
zwan018@ucr.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

ABSTRACT

Some recent papers showed that many sequential iterative algorithms can be directly parallelized, by identifying the *dependences* between the input objects. This approach yields many simple and practical parallel algorithms, but there are still challenges to achieve work-efficiency and high-parallelism. Work-efficiency means that the number of operations is asymptotically the same as the best sequential solution. This can be hard for certain problems where the number of dependences between objects is asymptotically more than optimal sequential work, and we cannot even afford the cost to generate them. To achieve high-parallelism, we always want it to process as many objects as possible in parallel. The goal is to achieve $\tilde{O}(D)$ span for a problem with the deepest dependence length D. We refer to this property as *round-efficiency*. This paper presents work-efficient and round-efficient algorithms for a variety of classic problems and propose general approaches to do so.

To efficiently parallelize many sequential iterative algorithms, we propose the *phase-parallel framework*. The framework assigns a *rank* to each object and processes the objects based on the order of their ranks. All objects with the same rank can be processed in parallel. To enable work-efficiency and high parallelism, we use two types of general techniques. Type 1 algorithms aim to use range queries to extract all objects with the same rank to avoid evaluating all the dependences. We discuss activity selection, and Dijkstra's algorithm using Type 1 framework. Type 2 algorithms aim to *wake up* an object when the last object it depends on is finished. We discuss activity selection, longest increasing subsequence (LIS), greedy maximal independent set (MIS), and many other algorithms using Type 2 framework.

All of our algorithms are (nearly) work-efficient and round-efficient, and some of them (e.g., LIS) are the first to achieve the both. Many of them improve the previous best bounds. Moreover, we implement many of them for experimental studies. On inputs with reasonable dependence depth, our algorithms are highly parallelized and significantly outperform their sequential counterparts.

CCS CONCEPTS

• **Theory of computation** → **Parallel algorithms**; **Shared memory algorithms**; **Algorithm design techniques**; *Shortest paths.*

KEYWORDS

parallel algorithms, phase-parallel framework, parallel programming, sequential iterative algorithms, activity selection, longest increasing subsequence, maximal independent set, independence system

ACM Reference Format:
Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. 2022. Many Sequential Iterative Algorithms Can Be Parallel and (Nearly) Work-efficient. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3490148.3538574

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07.
https://doi.org/10.1145/3490148.3538574

1 INTRODUCTION

There are two goals in designing efficient parallel algorithms: to reduce work, and to improve parallelism. *Work-efficiency*, meaning that the *work* (total number of operations) is asymptotically the same as the best sequential solution, is crucial for practical parallel algorithms. This is because nowadays and for the foreseeable future, the number of processors in a machine (up to thousands) is roughly polylogarithmic to input sizes. Hence, a parallel algorithm is less practical if it blows up the work of the best sequential algorithm by a polynomial factor. In this paper, we show (nearly) work-efficient parallel algorithms for a list of classic problems.

Our work is motivated by a list of recent papers that directly parallelize some sequential iterative algorithms (i.e., sequential algorithms that iteratively process input objects) [10, 12, 13, 16–18, 42, 47, 60, 64]. Their work-efficiency analysis usually directly follows the original sequential algorithm. To achieve high parallelism from a sequential iterative algorithm, the key is to identify the *dependences* among "objects" (e.g., iterations, instructions, or input objects), and process them in the proper order [10, 12, 13, 16–18, 42, 45, 47, 49, 60, 64]. In particular, we want to avoid waiting for "false dependences" and to process as many objects as possible in parallel. Such relationships can be modeled as a directed acyclic graph (DAG), referred to as the ***dependence graph*** (***DG***). Each object corresponds to a vertex in the DG, and a directed edge from vertex u to v means v can be executed only after u is finished, and we say v ***relies on*** (or depends on) u.

There are two existing general frameworks to design parallel algorithms using DGs, but they both have limitations. The first framework is *deterministic reservations* [10] (also used in [64]). These algorithms run in rounds, and in each round, check the unfinished objects, execute the "ready" objects in parallel, and postpone the rest to later rounds. This framework provides good parallelism—for a DG of depth D, we only need $O(D)$ rounds. In this paper, we define a computation as ***round-efficient*** if it executes a DG with depth D in $\tilde{O}(D)$ span (longest dependence of instructions in the

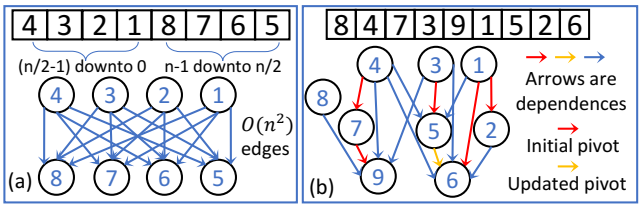


Figure 1: Examples of Longest Increasing Subsequence (LIS). (a) An example where we have $O(n^2)$ dependences between objects for input size n. (b) An example of how our algorithm processes LIS. Each object x chooses a pivot among its predecessors (red arrows). We check the readiness of x only when its pivot finishes. If x is still not ready, we update the pivot to another unfinished object. This avoids checking all edges in the DG.

algorithm, formally defined in Sec. 2)[1]. Despite the round-efficiency, deterministic reservations do not guarantee work-efficiency—the work in the worst case is $O(Dm)$, where m is the number of edges of the DG (using a topological sort sequentially only takes $O(m)$ work). The second approach was proposed by Blelloch et al. [12] to prove work-efficiency of DG-based algorithms, but it only applies to when each vertex in the DG has a constant in-degree. As a result, most of the existing algorithms do not fit in these two frameworks, and each [12, 16–18] uses specific design and analysis to get the work and span bounds (if any). Moreover, previous approaches are *edge-centric*—all edges in the DGs are examined to find ready vertices/objects. We observed that in many cases, even generating all edges in the DG is work-inefficient. Consider the classic longest increasing subsequence (LIS) problem that can be solved sequentially in $O(n \log n)$ work. In the worst case, the DG can contain $O(n^2)$ edges (see an example in Fig. 1). It remains open whether we can design work-efficient parallel algorithms with non-trivial parallelism for many classic problems such as LIS, and even general approaches to achieve so.

Contributions in this paper. We propose *a general framework and algorithms in this framework to parallelize sequential iterative algorithms, many of them textbook greedy and dynamic programming (DP) algorithms, that are (nearly) work-efficient and round-efficient.* Our approaches are *vertex-centric*, which avoid examining all edges in the dependence graph. We define the ***rank*** for each vertex in the DG (an input object or a subproblem), and we prove that rank fully captures the earliest "phase" that an object can be processed in a parallel algorithm. We believe that defining rank simplifies parallel algorithm design for many problems. Based on rank, we propose the ***phase-parallel algorithm framework***, which processes all objects based on the ordering of ranks, and round i processes all objects with rank i in parallel. For example, the rank of an object in the LIS problem is the LIS length ending at this object. The phase-parallel algorithm will then process all objects with LIS size i in round i, which finish in k rounds for an input sequence with LIS length k. We present the list of problems discussed in the paper, their ranks, and the cost bounds of our solutions in Tab. 1.

To achieve work-efficiency and round-efficiency, we propose two types of general ideas, referred to as Type 1 and Type 2 algorithms.

[1]We note that round-efficiency does not guarantee optimal span, since round-efficiency is with respect to a given DG. One can re-design a completely different algorithm that has a shallower DG and a better span.

Type 1 algorithms aim to identify objects to be processed in round i efficiently. To do this, we use *range queries* based on parallel augmented trees [66] (see Sec. 2), which take polylogarithmic work, instead of work proportional to the number of relevant edges in the DG. This idea applies to many greedy or DP algorithms, and our algorithms use range queries to find the maximal set of parallelizable objects. Type 1 algorithms include activity selection, and Dijkstra's algorithm (we discuss more in the full version [62] of this paper). We note that some of these ideas may have been known. We do not claim them as the main contribution, but use them to exemplify our framework.

Our Type 2 algorithms aim at *waking up* the ready objects at the right time. Instead of checking the readiness of all objects in each round (as was done in previous work), we hope to touch an object *only when it is (almost surely) ready*. One of our approaches is to assign a *pivot* p_x to each object x, which is an object x relies on. Only when p_x finishes, we check if x is ready. If x is ready, we process it in the next round. If not, we update x's pivot to another unfinished object it relies on. Fig. 1(b) shows an example of LIS. Although there are many dependences (all edges) in the DG, each object only picks one pivot (the red edges). When the pivot is ready, the object itself is likely to be ready, but if not (e.g., ⑥ first picks ①), it selects a new pivot (the yellow edge from ⑤). Therefore, only a small fraction of edges in the DG (the red and yellow ones) are evaluated, which saves work. Another approach is based on a new structure to identify a ready object when its last predecessor in DG finishes, which we apply to the greedy maximal independent set (MIS) algorithm and other similar algorithms. Our approach is based on a new data structure called *TAS tree*, which makes use of the atomic *test-and-set* operation (formally defined in Sec. 2). For MIS, our algorithm is work-efficient, and improves existing span bound [13] from $O(\log^3 n)$ to $O(\log^2 n)$.

We believe that our framework applies to a broad set of problems. We picked the examples by reviewing the problems in Cormen, Leiserson, Rivest, and Stein [30]. All algorithms in this paper are (nearly) work-efficient (only the LIS algorithm has an $O(\log^2 n)$ factor of overhead), and round-efficient.

Our framework applies to many existing algorithms in [10, 12, 13, 16–18, 42, 47, 60, 64], improves the bounds for many of them (e.g., the greedy MIS algorithm), and provides a much simpler way to understand these algorithms. As another example, our LIS algorithm (Algorithm 3) has $\tilde{O}(n)$ work and $\tilde{O}(k)$ span for LIS length k. Parallel LIS is widely studied [5, 43, 52, 53, 58, 59, 61, 67]. Our algorithm is the first to achieve near-work-efficiency and round-efficiency, which is advantageous especially for small output size. We review the literature of parallel LIS in Sec. 5. Our algorithm is also simpler and more practical. We are unaware of any implementations of these previous algorithms with competitive performance to the standard sequential LIS algorithm with $O(n \log n)$ work.

We implement many of these algorithms, and test them as a proof-of-concept to show how work- and round-efficiency affect practical performance. Although there is parallel overhead and our worst-case span is $\tilde{O}(n)$, our work-efficient algorithms achieve significant speedup over the sequential algorithm in a reasonably large input parameter space. Our contributions include:

- The phase-parallel framework to parallelize sequential iterative algorithms based on the concept of *rank* defined in this paper.
- Two general techniques for phase-parallel algorithms to achieve work-efficiency and round-efficiency, based on range queries and a wake-up strategy to identify ready objects, respectively.
- The first **nearly work-efficient** ($\tilde{O}(n)$ work) LIS algorithm with **round-efficiency** ($\tilde{O}(k)$ for LIS length k) and its implementation.
- A **work-efficient** greedy maximal independent set (MIS) algorithm with $\boldsymbol{O(\log n \log d_{max})}$ **span** *whp* in the binary-forking model (defined in Sec. 2), where n is the number of vertices and d_{max} is the maximum degree in the graph. This improves the previous best span bound of $O(\log^3 n)$.
- Two algorithms for the activity selection problem. This is the first parallel algorithm for this problem we know of. Although not complicated, they reveal the connections of the two types of algorithms in this paper. They are work-efficient and round-efficient. For the unweighted version, we also provide an algorithm with $O(\log n)$ span *whp*.
- Many other simple and interesting algorithms using our framework, including Huffman tree, SSSP, unlimited knapsack, etc.
- Implementations and experimental studies of these algorithms.

2 PRELIMINARIES

Notations. For a sequence s, s_i or $s[i]$ denotes the i-th element, and $s_{i\ldots j}$ or $s[i\ldots j]$ denotes the i-th to the j-th elements in s. We use the term $O(f(n))$ ***with high probability*** (*whp*) in n to indicate the bound $O(kf(n))$ holds with probability at least $1-1/n^k$ for any $k \geq 1$. With clear context we drop "in n".

Longest Increasing Subsequence (LIS). Given a sequence $s_{1\ldots n}$, $s'_{1\ldots m}$ is a subsequence of s if $s'_i = s_{k_i}$, where $k_1 < k_2 < \ldots k_m$. Given a sequence s, the ***longest increasing subsequence*** (LIS) problem finds the longest subsequence s^* of s where $\forall i, s^*_i < s^*_{i+1}$.

Dependence Graph. A sequential iterative algorithm processes each input object in a given order. The dependence graph (DG) represents the processing dependence of input objects. Each vertex in the DG denotes an input object. An edge from x to y means that object y can be processed only when x has been finished, and we say y ***relies on*** x. We say an object is ***finished*** if it has been processed, and ***unfinished*** otherwise. We say an object is ***ready*** if all its predecessors in the DG have been finished. For two objects x and y, where y relies on x and x is unfinished, we say x ***blocks*** y.

Parallel Computational Model. We use the work-span model on the binary-forking model (with `test_and_set`) to analyze parallel algorithms [12, 30] as is used in many recent papers [2, 4, 6–8, 11, 14–16, 18–21, 25–27, 36, 38, 46]. We assume a set of threads that share a memory. Each thread supports standard RAM instructions and a `fork` instruction that forks two new child threads. When a thread performs a `fork`, the two child threads both start by running the next instruction, and the original thread is suspended until both children terminate. A computation starts with a single root thread and finishes when that root thread finishes. We use atomic operation `test_and_set` (TAS), which checks whether a memory location is of zero, sets it to one if so, and return its old value. We say a TAS is ***successful*** if it changes zero to one and ***unsuccessful*** otherwise. One can use TAS to implement a *join* operation in the standard fork-join model [12]. An algorithm's ***work*** is the total number of instructions, and the ***span*** (depth) is the length of the longest sequence of dependent instructions in the computation. Note that a parallel for-loop incurs $O(\log n)$ span because of binary-forking. We can execute the computation efficiently using a randomized work-stealing scheduler both in theory and in practice [12, 22, 30].

Parallel Data Structures. We now present useful theorems of data structures for range-sum queries used in this paper. More details are given in the full version of this paper [62]. The data structures maintain a map of entries (*key-values*) sorted by the keys, where keys can be either one or two dimensions. A range sum query is defined by a key range (an interval in 1D or a rectangle in 2D) and *augmentation*, which defines how the "sum", called the *augmented value*, should be computed. One example is to report the sum (or min/max) of values in the given key range. Formally, suppose the map maintains key-value pairs of type $K \times V$, we define the augmented value of type A by an augmented structure consisting of two functions and the identity of A.

- Base function $g : K \times V \mapsto A$, which maps an entry (key-value) to an augmented value.
- (Associative) Combine function $f : A \times A \mapsto A$, which combines (adds) two augmented values into a new augmented value.
- The identity $I_A \in A$ of f on A. (A, f, I_A) is a monoid.

In the value-sum example above, the base function $g = (k, v) \mapsto v$, the combine function $f = (a_1, a_2) \mapsto a_1 + a_2$, and $I_A = 0$. We first present a useful theorem about range sum queries.

Theorem 2.1. *For $k \in \{1, 2\}$, there exist data structures that can answer k-D range sum query in $O(\log^k n)$ time, can be constructed by (or be flattened into) a sorted sequence of entries in $O(n \log^{k-1} n)$ work and $O(\log^k n)$ span, and allow for batch update (e.g., insertion, deletion and value updates) in $O(m \log^k n)$ work and $O(\log^k n)$ span, where n is the number of input entries, and $m \leq n$ is the batch size. We assume constant cost for the base and combine functions.*

This can be achieved using parallel augmented balanced binary search trees (PA-BST) [66] with algorithms in [9, 12, 34, 66]. For 2D range sum queries, we use a 2D range tree using PA-BSTs [65, 66].

We also use multi-map, where multiple entries can have the same key, and a search on a key will return all values with this key (Note that this query is different from the 1D or 2D range query defined above, so this uses a different data structure from range trees stated above). By using PA-BST, we have the following theorem [9].

Theorem 2.2. *For n key-values, there exists a data structure that can search or update (insert or delete) a batch of entries in $O(m \log n)$ work and $O(\log m \log n)$ span, where $m \leq n$ is the total number of elements found or updated in the batch.*

3 PHASE-PARALLEL ALGORITHMS

In this section, we introduce our key concept: ***phase-parallel algorithms***, and show a general approach to design phase-parallel algorithms to maximize parallelism based on the ***rank*** function. Since our idea is sophisticated, we first show the pseudocode in Algorithm 1 and describe the high-level idea.

To seek parallelism in many sequential iterative algorithms, we define the $rank(\cdot)$ of each object to capture the dependences among

	Feasible Condition for objects x and y	$rank(x)$	Type	Work	Span
Activity Selection (general)	x and y do not overlap	The maximum number of non-overlapping activities ending at x	1&2	$O(n \log n)$	$O(rank(S) \log n)$
Unlimited Knapsack	Solution to weight y can contain solution to the subproblem of weight x	x/w^*, where w^* is the minimum weight	1	$O(Wn)$	$O(rank(S) \log(w^* n))$
Huffman Tree	y's Huffman code is a prefix of x	Subtree height of x	1	$O(n \log n)$	$O(rank(S) \log n)$
Dijkstra's Algorithm	x is on the shortest path to y	Hop distance from x to the source on shortest path tree $\overline{rank}(x) = d(x)/\min_{e \in E} w(e)$	1	$O(m \log n)$	$O\left(\frac{\max_{v \in V} d(v)}{\min_{e \in E} w(e)} \log n\right)$
LIS	$y > x$	The length of LIS ending at x	2	$O(n \log^3 n)^\dagger$	$O(rank(S) \log^2 n)^\dagger$
Activity Selection (unweighted)	x and y do not overlap	The maximum number of non-overlapping activities ending at x	2	$O(n \log n)$	$O(\log n)^\dagger$
MIS	$\exists$ a path from x to y s.t. the priorities on the path are monotonically increasing	The the longest chain size ending at vertex x with increasing priorities	2	$O(n + m)$	$O(\log^2 n)^\dagger$

Table 1: The problems, their definitions of rank, the work and span of our solutions for the given problems. In feasible conditions, we assume y is later than x. $n = |S|$ is the input size. For graphs, n is the number of vertices and m is the number of edges. $d(v)$ in SSSP is the shortest distance of v from the source and $w(e)$ is the weight of edge e. W in the knapsack problem is the weight limit. $\overline{rank}$ means a relaxed rank (see Sec. 4.2). †: with high probability.

them, which indicates the earliest phase an object can be ready. With a properly defined rank function, Algorithm 1 processes all objects (in parallel) of rank i in round i. We call the set of objects processed in round i (T_i in Algorithm 1) the ***frontier*** of round i. Table 1 shows how rank is defined for the problems in this paper. For example, in the LIS problem, an object's rank is the size of the LIS ending at this object. Therefore, Algorithm 1 finds and processes all objects with LIS size 1 in parallel, then those with LIS size 2, etc. Throughout the section, we use the LIS problem as an example to help understand the abstract concepts. Next, we formalize the phase-parallel algorithms. We note that all of our algorithms are reasonably simple. The goal of the formalization is to extend our idea to general independence systems, which generalizes to more DP and greedy algorithms.

Algorithm 1: The phase-parallel algorithm

```
Input: S, and rank(x) that implies F
i ← 1
while S ≠ ∅ do
    Find the set T_i that contains all objects with rank i
    Process all objects in T_i in parallel
    S ← S \ T_i
    Update the status of objects in S if necessary
    i ← i + 1
```

An independence system is a pair $(S, \mathcal{F})$, where S is a finite set and $\mathcal{F}$ is a collection of subsets of S (called the ***independent sets*** or ***feasible sets***) with the following properties:

(1) The empty set is feasible, i.e., $\emptyset \in \mathcal{F}$.
(2) (Hereditary property) A subset of a feasible set is feasible, i.e., for each $Y \subseteq X$, we have $X \in \mathcal{F} \implies Y \in \mathcal{F}$.

A feasible set for the LIS problem is any increasing subsequence.

Given an independence system $(S, \mathcal{F})$, a ***sequential order*** of it is a permutation of all objects in S, usually specified by the input. For an object $x \in S$, let $\mathcal{I}_S(x)$ be the index of x w.r.t. its sequential order. We say an object x is ***earlier*** than y if $\mathcal{I}_S(x) < \mathcal{I}_S(y)$, and ***later*** otherwise. Let $x^{\downarrow S} = \{y \in S : \mathcal{I}_S(y) \le \mathcal{I}_S(x)\}$ be the downward closure of x, i.e., all objects no later than x. With clear context, we drop the superscripts and use $\mathcal{I}(x)$ and $x^{\downarrow}$. We use S_i as the object in S with index i. In LIS, the index $\mathcal{I}(x)$ of an object x is its position in the input sequence S, and $x^{\downarrow}$ is the prefix of S up to x.

We say two objects x and y are ***incompatible*** if $\nexists E \in \mathcal{F}$, s.t. $x \in E$ and $y \in E$, and ***compatible*** otherwise. We say an object x is compatible with a set $E \subseteq S$ if $E \cup \{x\} \in \mathcal{F}$. Mapping this to LIS, two objects x and y (later than x) are compatible iff $x < y$.

Given an object x, we use $\mathcal{F}(x) = \{E \in \mathcal{F} : E \subseteq x^{\downarrow}, x \in E\}$ to denote all feasible sets *with the last object as* x, and the ***Maximum Feasible Set (MFS)***[2] $MFS(x) = \arg\max_{E \in \mathcal{F}(x)} |E|$ as the largest set among $\mathcal{F}(x)$. For many DP problems, MFS is usually related to the *DP value* of the object. For example, in LIS, the $MFS(x)$ is the LIS ending at $x \in S$. For a set S, we also define the MFS as the largest feasible subset of S. We define the ***rank*** of a set or an object as $rank(\cdot) = |MFS(\cdot)|$. In LIS, $\mathcal{F}(x)$ refers to all increasing subsequences ending at object x. The MFS of an input sequence is the LIS of the sequence. $rank(x)$ is the size of LIS ending with x.

Given an independence system $(S, \mathcal{F})$, a ***sequential iterative algorithm*** $\mathcal{A}$ on S processes each object S_i in S iteratively based on the sequential order, with the goal to optimize some value of all (or some) feasible sets. Since a processed object usually corresponds to a subproblem on $S_i^{\downarrow}$, they are sometimes called the ***states*** in dynamic programming problems. For example, in LIS, processing object S_i is to compute the LIS up to (and including) object S_i.

To parallelize a sequential iterative algorithm, note that an object does not need to wait for *all* earlier objects to finish, but only a subset of them. Let $\mathcal{P}(x)$ be all objects that x rely on, i.e., all predecessors of x in the DG. For LIS, $\mathcal{P}(x) = \{y : \mathcal{I}(y) < \mathcal{I}(x), y < x\}$. When all objects in $\mathcal{P}(x)$ finish, x is ready. Also, if two objects do not rely on each other in the DG, they can be processed in parallel. These two simple observations have been used in existing parallel algorithms and frameworks (e.g., [13, 16, 16–18, 47, 64]). In this paper, we formalize the problem for a class of algorithms based on an independence system and point out that identifying the ready objects can be captured by the *ranks* of the objects. We define *phase-parallel* as follows.

Definition 3.1. Given an independence system $(S, \mathcal{F})$, a sequential iterative algorithm on S is ***phase-parallel*** if it has the following

[2]This is also known as the maximum independent set (MIS). In this paper, to avoid confusion with the greedy MIS algorithm in Sec. 5.3, we use the term MFS.

property: an object $x \in S$ relies on $y \in S$ in the parallel dependence graph if and only if:

(1). (Ordering) $\mathcal{I}(y) < \mathcal{I}(x)$.
(2). (Compatibility) $\forall E \subseteq \mathcal{F}(y)$, we have $E \cup \{x\} \in \mathcal{F}$, i.e., any feasible set containing y and only objects up to y are also compatible with x.

This means computing the state (processing an object) x only relies on previous states in $x^{\downarrow}$ compatible with x. This indicates *optimal substructure* property [30], where the best solution at x can be obtained by optimal solutions before x.

To achieve maximum parallelism, our goal is to find the largest possible set of objects to process in parallel. We first show that all objects with the same rank (MFS size) can be processed in parallel.

Theorem 3.2. *Given a phase-parallel algorithm $\mathcal{A}$ on the independence system $(S, \mathcal{F})$, if $rank(x) = rank(y)$, then x and y cannot rely on each other in the parallel dependence graph.*

Proof. Assume to the contrary that y relies on x (the other case is symmetric). Consider the MFS of y. By definition $MFS(y) \cup \{x\}$ is also feasible, which means $rank(x) \geq rank(y) + 1$. □

Thm. 3.2 leads to the following conclusion, based on which we propose Algorithm 1.

Corollary 3.3. *In a dependence graph, if x relies on y, $rank(x) > rank(y)$. All objects with the same rank can be processed in parallel.*

Theorem 3.4. *The rank of an object in a phase-parallel algorithm is its depth in the DG.*

Proof. (Sketch) From the compatibility property of the phase-parallel algorithm, we know the rank (MFS size) of an object must be 1 plus the maximum MFS size of its predecessors. By induction, we can prove the given theorem. □

Theorem 3.4 verifies the strategy of Algorithm 1, which means we are just processing the objects based on their depth in the DG.

In this paper, we propose novel and efficient ways to process phase-parallel algorithms. The challenges lie in achieving work-efficiency with non-trivial parallelism. As mentioned, although the high-level idea of processing all ready objects (thus achieving round-efficiency) in a round has been used in existing work, most of them need to check all edges in the DG. In many cases, the number of edges can be asymptotically more than efficient work. We propose two general ideas to reduce work in phase-parallel algorithms. Type 1 algorithms (Sec. 4) find the frontier in each round efficiently using a *range query* in polylogarithmic cost. Type 2 algorithms (Sec. 5) *wake up* all ready objects by the finished ones at the right time. Both cases avoid checking all edges in the DG.

4 TYPE 1 ALGORITHMS USING EFFICIENT FRONTIER IDENTIFYING

Type 1 algorithms exhibit the property where each object maintains a value, and all objects with the same rank have their values in a contiguous range. We will use PA-BST to maintain these values and use a range search to efficiently find the frontier. We show activity selection and Dijkstra's algorithm in this paper, and more (unlimited knapsack and Huffman tree) in the full version [62]. Many of them are straightforward. We do not claim all of them as the main contributions, but use them as simple examples to understand our framework.

4.1 Activity Selection

Activity selection is a textbook example of greedy or dynamic programming algorithms [30]. Given a set of activities $S = \{A_i\}$ defined by their start time s_i, end time e_i and weight w_i, the problem is to find a feasible subset of non-overlapping activities to maximize the total weight. When all activities have a unit weight, a simple *earliest-end* greedy strategy can solve the problem [30], i.e., repeatedly selecting the earliest ending activity and removing all incompatible (overlapped) activities. The general version (arbitrary weight) can be solved by the dynamic programming recurrence:

$$dp[i] = \max_{e_j \leq s_i} dp[j] + w_i \tag{1}$$

The sequential order (and the *index*) is defined by the end time. We assume all activities are pre-sorted by their end time. A feasible set is a set of non-overlapping activities. $dp[i]$, or the ***DP value*** of activity i, means the highest possible weight by using the first i activities, which must include A_i. Naively computing Eq. (1) needs $O(n^2)$ work. Since the condition in Eq. (1) is a range of end time, sequentially, the work can be reduced to $O(n \log n)$ using augmented range queries. In parallel, an activity A_i depends on all activities ending before A_i starts (see an illustration in Fig. 2), which leads to $O(n^2)$ dependences in the worst case. Note that the rank of A_i by definition is the maximum size of the feasible set containing A_i and only the first i activities. We first present the following lemma.

Lemma 4.1. *All activities overlapping the earliest-end activity A_1 has rank 1. After removing all activities with rank no more than k, suppose the earliest-end activity is A_j, then all remaining activities that overlaps with A_j has rank $k + 1$.*

Proof. We first prove all activities overlapping the earliest-end activity A_1 must have rank 1. Recall the rank of an activity x is the maximum number of compatible activities we can select from x and earlier activities, which must include x. For an activity overlapping A_1, if its rank is larger than 1, there must exist another activity y that is earlier than x and compatible with x. This means y ends before x starts, which contradicts the assumption that the earliest-end activity overlaps with x.

We next prove that after removing all activities of rank no more than k, if the current earliest-end activity is A_j, all activities overlapping A_j should have rank $k + 1$. Let S' be the set of activities removed, which have rank no more than k. Assume to the contrary that one of such activity A_i has rank $r > k + 1$. Then there must be an activity $A_{i'}$ finishing earlier than A_i, and have rank $r' > k$. Therefore, $A_{i'} \notin S'$ since we only remove activities with rank no more than k. However, this contradicts the assumption that A_j is the earliest-end activity in $S \backslash S'$, and A_j already overlaps A_i. □

Based on the lemma and the phase-parallel framework, we can design an algorithm (Algorithm 2). To enable work-efficiency, we use a range query to find the largest parallelizable frontier. The algorithm uses two PA-BSTs T_{time} and T_{DP}. T_{time} maintains all unprocessed activities sorted by their start time and augmented on the minimum end time, which is used to identify the frontiers. T_{DP}

Algorithm 2: Type-1 activity selection algorithm

Input: All activities' start time s_i, end time e_i and weight w_i
1 Build PA-BSTs T_{time} on key-values (s_i, e_i), augmented on the minimum end time, and T_{DP} on key-values $(e_i, dp[i])$, augmented on the maximum DP value
2 **while** $T_{time} \neq \emptyset$ **do**
3 Find the earliest-end activity x by the augmented value of T_{time}
4 $\langle T, T' \rangle \leftarrow split(T_{time}, e_x)$ `// All activities starting before` e_x `form the current frontier T`
5 **parallel_for_each** activity $i \in T$ **do**
6 $dp[i] = w_i + T_{DP}.range(-\infty, s_i)$
7 Update all DP values of activities in T in T_{DP} in parallel
8 $T_{time} \leftarrow T'$ `// remove finished objects`

maintains all activities sorted by their end time and augmented on the largest DP value, which is used to determine $\max_{e_j \le s_i} dp[j]$ in the DP recurrence. In each round, we find the earliest-end activity x by reading the augmented value of T_{time}. Then we split T_{time} based on e_x. Those starting no later than e_x will be split out as the frontier and will be processed in parallel. Since T_{time} is indexed on start time, Split takes $O(\log n)$ work. When processing activity i, we use T_{DP} to extract the highest DP value among all activities with end time in range $(-\infty, s_i]$, and use it to update the DP value of i in T_{DP}. The work for processing m objects in the frontier is $O(m \log n)$ for the augmented range query, and $O(m \log n)$ for updating the DP values in T_{DP}. This leads to the following theorem.

Theorem 4.2. *Type 1 activity selection algorithm takes $O(n \log n)$ work and $O(rank(S) \log n)$ span, where S is the input set and $n = |S|$.*

4.2 Algorithms with Relaxed Rank

In some problems, it is hard to find (or use) the exact rank of the objects, in which case we use a relaxed rank, defined as follows.

Definition 4.3. Given a phase-parallel algorithm $\mathcal{A}$ on the independence system $(S, \mathcal{F})$, a function $\overline{rank}(x)$ on $x \in S$ is a ***relaxed rank*** on object x if

- $\forall x \in S, rank(x) \le \overline{rank}(x)$.
- For $x, y \in S$ where x relies on y in the dependence graph, $\overline{rank}(x) > \overline{rank}(y)$.

Then Algorithm 1 can process all objects with the same relaxed rank in each round. Note that the trivial relaxed rank is the index of each object $\mathcal{I}(x)$, which gives no parallelism in Algorithm 1. Therefore, when we use the relaxed rank, we need careful analysis to show non-trivial parallelism.

Dijkstra's Algorithm. Dijkstra's algorithm [37] solves the single-source shortest paths (SSSP) problem on a weighted graph. As a sequential iterative algorithm, Dijkstra processes the vertices in the order of their distances to the source and relaxes their neighbors. SSSP is very challenging in the parallel setting. Dijkstra is work-efficient and relaxes each edge exactly once. However, it is hard to parallelize because each round only processes one vertex. Bellman-Ford has better parallelism but significantly more work. Almost all state-of-the-art parallel SSSP algorithms (e.g., Δ-stepping [57] and ρ-stepping [39]) achieve high parallelism by using more work.

The DG of the SSSP problem is conceptually the shortest path tree. The rank of a vertex v is the hop distance from v to s in the shortest path tree. However, the algorithm itself is unaware of the explicit structure of DG before the shortest paths are computed. Hence, the exact rank of each object is hard to acquire. Let w^* be the smallest edge weight in the graph, and $d(v)$ the actual distance of v. We define a relaxed rank for a vertex $\overline{rank}(v) = \lceil d(v)/w^* \rceil$. This is because distances within a window of w^* cannot rely on each other (relaxation increases the distance by at least w^*). Therefore, each frontier can be extracted using a range query. Interestingly, we observe that this is (conceptually) similar to using $\Delta = w^*$ in Δ-stepping [57]. Using PA-BST to maintain the distances of all vertices, we have the following result.

Theorem 4.4. *There exists a parallel algorithm that solves SSSP problem on a graph $G = (V, E)$ using $O(|E| \log |V|)$ work and $O(\overline{rank}(V) \log |V|)$ span, where $\overline{rank}(V)$ is the ratio of the maximum shortest path in the graph and the smallest edge weight.*

We note that there can be other ways to define the relaxed rank of the Dijkstra's algorithm [31, 50], which enable different bounds to the phase-parallel algorithms. In the paper we simply discuss the version based on the smallest edge weight, since it can be easily tested using a Δ-stepping-based implementation.

In our experiments in Sec. 6, we use the Δ-stepping implementation in [39] with $\Delta = w^*$ to test the performance of this idea. On low-diameter graphs with reasonably large w^*, setting $\Delta = w^*$ gives the best performance among all choices of parameter Δ because of the work-efficiency.

5 TYPE 2 ALGORITHMS WITH WAKE-UP STRATEGIES

In phase-parallel algorithms, an object is ready when all its predecessors $\mathcal{P}(x)$ finish. Previous approaches require explicitly generating $\mathcal{P}(x)$ for each $x \in S$ [12, 16–18, 64] (achieving work-efficiency only when $|\mathcal{P}(x)| = O(1)$), checking the readiness of all objects every round [10] (not necessarily work-efficient), or based on dual-binary search [13, 47] (incurs overhead in span). To avoid exhaustedly checking the readiness of every object, Type 2 algorithms aim to *wake up* an object x when the *last* object in $\mathcal{P}(x)$ finishes.

We propose two wake-up strategies. Our first approach, which we believe is very interesting, is to avoid explicitly generating $\mathcal{P}(x)$, and check the readiness of an object x when x is likely to be ready. To do so, we attach each object x to an unfinished object $p_x \in \mathcal{P}(x)$, called the *pivot*, which blocks x. We redo the check only when p_x is ready, which bounds the number of total checks to be $O(\log |\mathcal{P}(x)|)$ *whp*. We show activity selection and longest increasing sequence (LIS) as examples, and more applications in the full version.

The second approach applies to algorithms that can afford to generate $\mathcal{P}(x)$ for each $x \in S$. Our idea is to build an asynchronous structure using `test_and_set` to precisely identify when the last object in $\mathcal{P}(x)$ is ready, and achieve better span. We present the greedy MIS algorithm as an example in Sec. 5.3, and more discussions in the full version of the paper.

5.1 Activity Selection

We now revisit the activity selection problem and present an algorithm using Type 2 framework. Recall the DP recurrence $dp[i] = \max_{e_j \le s_i} dp[j] + w_i$. Therefore, A_i is ready when all other activities

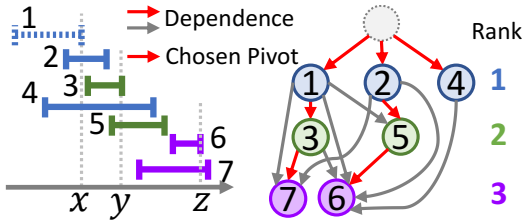


Figure 2: Illustration of the activity selection problem. Left: The start and end time of 7 activities (ordered by end time). Right: The dependences between activities. Rank 1 activities start before x (shown in blue). Rank 2 activities start before y (shown in green). Rank 3 activities start before z (shown in purple). Red dependences are the pivots chosen in the Type 2 algorithm. The pivot of an object is the compatible activity before it with the latest start time. A rank-r object has a pivot with rank $r-1$.

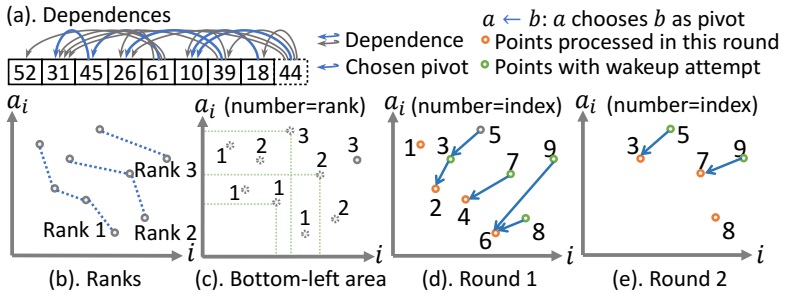


Figure 3: Illustration of the LIS algorithm. (a). Objects, their dependences, and the random pivot chosen. (b). Ranks of the objects represented as 2D points (i, a_i). (c). Objects (points) and their bottom-left area. A point with rank i only has points with rank $< i$ in its bottom-left area. (d). Points processed (①, ②, ④, ⑥) and waking-up attempts in round 1. ② wakes up ③. ④ wakes up ⑦. ⑥ attempts to wake up ⑧ and ⑨, but ⑨ is not ready. (e). Points processed (③, ⑦, ⑧) and waking-up attempts in round 2.

with end time before s_i have been processed. Our idea is to let each activity x find a *pivot* p_x, where finishing processing p_x indicates the readiness of x. We prove the following lemma.

Lemma 5.1. *Given activity A_x, let activity $A_{p_x} = \arg\max_{A_i: e_i \le s_x} s_i$ be the latest-start activity among all activities ending before A_x starts (i.e., those earlier than A_x and compatible with A_x), and call A_{p_x} the* **pivot activity** *of A_x. Then $rank(A_x) = rank(A_{p_x}) + 1$.*

Proof. By definition, A_x depends on A_{p_x} because A_x is compatible with all MFS ending with A_{p_x}. This indicates $|MFS(A_{p_x})| = rank(A_{p_x}) < rank(A_x) = |MFS(A_x)|$. We then show $MFS(A_{p_x}) \cup \{A_x\}$ is an MFS of A_x. Assume to the contrary that $t = |MFS(A_x)| > |MFS(A_{p_x})| + 1$. Consider such an MFS $T = MFS(A_x)$ where $|T| = t$. Let activity A_k be the activity with the latest starting time in $T - \{A_x\}$. We first prove $A_{p_x} \notin T - \{A_x, A_k\}$. This is because if $A_{p_x} \in T - \{A_x, A_k\}$, A_{p_x} should have an earlier starting time than A_k (by definition of A_k), which contradicts the definition of A_{p_x}. Similarly, all other activities in $T - \{A_x\}$ have earlier end time than s_k. By the definition of A_{p_x}, A_k starts no later than A_{p_x} (A_{p_x} is the latest starting activity before A_x and compatible with A_x). This means that A_{p_x} is also compatible with (and later than) $T - \{A_x, A_k\}$. $MFS(A_{p_x}) \ge |(T - \{A_x, A_k\}) \cup \{A_{p_x}\}| = t - 1$, contradicting the assumption of $t = |MFS(A_x)| > |MFS(A_{p_x})| + 1$. □

We show an example of pivot activities in Fig. 2. Lem. 5.1 implies that the pivot activity p_x of any activity x must be processed in the previous round of when x is processed. In other words, once p_x is finished, we can wake up x and process it in the next round. In this case, we can first let all activities find their pivot via binary searches, which is $O(\log n)$ work per activity. We use a tree T_{pivot} as a multi-map to store all pairs (p_x, x). We start with processing all activities with rank 1. For each activity y in the current frontier, after processing them, we find all pairs $(y, z) \in T_{pivot}$ and put all such z in the next frontier (they will be wakened up). An activity can be processed (computing its DP value) similarly as in Type 1 by using a PA-BST T_{DP}. We have the following theorem.

Theorem 5.2. *Type 2 activity selection algorithm takes $O(n \log n)$ work and $O(rank(S) \log n)$ span, where S is the input set and $n = |S|$.*

An $O(\log n)$ span algorithm for unweighted activity selection. Based on Lem. 5.1, we can further design a parallel algorithm with better span for the *unweighted* activity selection problem, where each activity has a unit weight ($w_i = 1$). Note that this is equivalent to computing the *rank* of each activity. Based on Lem. 5.1, we can rewrite the DP recurrence for the unweighted version as:

$$dp[i] = dp[j] + 1 : A_j \text{ is the pivot activity of } A_i$$

This simplifies the dependence graph to a tree structure, where each activity only relies on its pivot. The rank of each activity is also its depth in this tree, which can be computed using a standard tree contraction [18] in $O(n)$ work and $O(\log n)$ span *whp*.

Theorem 5.3. *The unweighted activity selection problem can be solved in $O(n \log n)$ work and $O(\log n)$ span* whp.

5.2 Longest Increasing Subsequence (LIS)

We propose a parallel algorithm for the longest increasing subsequence (LIS) problem using our phase-parallel framework. Given a sequence a, LIS asks for the longest subsequence in a that is strictly increasing. LIS is widely studied, and its parallel solutions have been studied in [5, 43, 48, 52, 53, 58, 59, 61, 67]. Most of these algorithms [43, 52, 58, 59, 61, 67] introduced polynomial overhead in work, and Alem and Rahman's algorithm [5] has $\tilde{\Theta}(n)$ span. Krusche and Tiskin's BSP algorithm [53] translates to $\tilde{O}(n)$ work and $\tilde{O}(n^{2/3})$ span, which is the only nearly-work-efficient algorithm with sublinear span. This algorithm relies on complicated techniques from [68], and has no implementation. In fact, we are unaware of any previous parallel LIS implementation with competitive performance to the standard sequential $O(n \log n)$ LIS algorithm.

Sequentially, LIS can be computed using the dynamic programming (DP) recurrence as follows. Let $dp[i]$, called the DP value, be the LIS length of $a_{1..i}$ ending with a_i. Then

$$dp[i] = \max(1, \max_{j<i, a_j<a_i} dp[j] + 1) \quad (2)$$

We can iteratively compute $dp[i]$ and maintain any search structure to find $\max_{j<i, a_j<a_i} dp[j]$ in $O(\log n)$ work. Our phase-parallel LIS algorithm parallelizes this sequential algorithm and achieves nearly work-efficiency and round-efficiency. Moreover, our algorithm is implementable and we show experimental study in Sec. 6.3. Here we maximize LIS length, but our algorithm can be generalized to the weighted case where objects have different weights.

An object a_i is ready once all objects a_j with $j < i, a_j < a_i$ are ready. In our phase-parallel framework, the rank of an object a_i is the size of the LIS ending with a_i (its DP value). An object a_i only depends on objects with a smaller rank. After all objects with rank r have finished, all objects with rank $r+1$ must be ready. The main challenge is to avoid processing all dependencies since there can be $\Theta(n^2)$ of them. Interestingly, the problem exhibits a nice geometric property. If we draw each object as a 2D coordinate (i, a_i), all the objects that a_i relies on are the points to its lower-left area (see Fig. 3). Therefore we can determine if an object x is ready using a 2D range query on the number of unfinished objects to its lower-left area. Similarly, the DP value of x can be obtained by querying the maximum DP value among all objects in its lower-left area (and plus one). Both queries can be done by a augmented 2D range tree (see Sec. 2). This gives a simple algorithm, where in each round, we can run a 2D range query to every unfinished object to identify the ready ones, and then compute their DP values. However, this can still incur $\Omega(n^2)$ work in the worst case.

To achieve work-efficiency, we would like to wake up an object only when it is almost ready. Unlike activity selection, we found it hard to find an *exact* pivot for each object x that has rank $rank(x)-1$. Instead, our strategy is to randomly pick an unfinished object in $\mathcal{P}(x)$ as x's pivot, which can be efficiently supported by an augmented 2D range tree. When the pivot of x is processed, we *attempt* to wake up x by checking whether all objects in $\mathcal{P}(x)$ (those in its lower-left corner) are finished. If so, x is ready, and we query the maximum DP value in $\mathcal{P}(x)$ to compute x's DP value. Otherwise, x is not waked up successfully, and selects another random unfinished object in its lower-left corner as the new pivot. In each round, all ready objects will attempt to wake up all objects using x as the pivot. This inductively guarantees that objects with rank i (LIS length i) are waked up and processed in round i.

Our algorithm is in Algorithm 3. Each object corresponds to a *point*, defined on its x-coordinate (its index i), and y-coordinate (a_i). We also maintain its DP value dp (initialized to $+\infty$). We create a virtual point $p[0]$ as a starting point with index 0 and value $-\infty$.

We use a range tree T_{range} to maintain all the points in the 2D planar, augmenting on a triple $\langle n_\infty, dp^*, x^*\rangle$, which records for the current subtree, the number of unfinished points n_∞, the maximum DP value dp^*, and an x-coordinate x^* (an index). If the maximum DP value dp^* is ∞, which means that there exist unfinished elements in this subtree, then the index x^* is selected uniformly at random from the unfinished objects. Otherwise, x^* is used to record the index to achieve the maximum DP value, which can be used to reconstruct the LIS if needed. To maintain such augmented values, the combine function simply adds up n_∞ (Lines 18 and 19), and takes a maximum on dp^* (Line 15) on the two augmented values a_1 and a_2. If dp^* is not $+\infty$, x^* can be simply set to be the argument to achieve the highest DP value (Line 19). Otherwise, x^* is selected from the x^* value of either a_1 or a_2, and the probability is decided by $t_1 : t_2$, where t_1 and t_2 are the number of unfinished objects (the n_∞ values) in a_1 and a_2, respectively (Line 17). By doing this, x^* is selected uniformly at random from both a_1 and a_2. We also use a multi-map T_{pivot} to maintain the pivot-object pairs.

The algorithm starts from a frontier of the virtual point $p[0]$. Initially, T_{pivot} stores $(0, i)$ as the initial pivot for all i. In each round, the algorithm processes each object x in the frontier in parallel.

Algorithm 3: The parallel LIS algorithm

Input: A sequence $a[1..n]$ with comparison function $<$
Output: The LIS length of $a[\cdot]$

```
1  Struct Point contains
2      int x, y                                  // x = index, y = a[x]
3      int dp                                    // The DP value
4  Point p[1..n]
5  p[0] ← ⟨0, −∞, 0⟩                            // insert virtual point 0
6  parallel_for_each a[i] ∈ a do  p[i] = ⟨i, a[i], +∞⟩
7  RangeTree<Point> T_range with
8      <_x (p1, p2): return p1.x < p2.x
9      <_y (p1, p2): return p1.y < p2.y
       // n_∞: # of unfinished points (dp value ∞)
       // dp*: max dp value in subtree
       // x*: index (x) of max dp value
10     augmented value: ⟨n_∞, dp*, x*⟩
11     base(p):
12         if p.dp = +∞ then return ⟨1, p.x, p.dp⟩
13         else return ⟨0, p.x, p.dp⟩
14     combine(a1, a2):                          // combine two aug values
15         m ← arg max_{i∈{1,2}} a_i.dp*
           /* When max dp value is +∞, choose a uniformly
              random one as the potential pivot          */
16         if a_m.dp* = +∞ then
17             t ←random(1, 2) with probability a1.n_∞ : a2.n_∞
18             return ⟨a1.n_∞ + a2.n_∞, +∞, a_t.x*⟩
19         return ⟨0, a_m.dp*, a_m.x⟩
20 Construct T_range from p[·]
21 Multi-map T_pivot = {(0, i) : i = 1..n}   // pivot pairs (p_x, x)
22 frontier = {0}
23 while frontier ≠ ∅ do
24     frontier = WakeUp(frontier)
25 return max_i(p[i].dp)

26 Function WakeUp(frontier)
27     todo ← T_pivot.multi_find(frontier)
28     parallel_for_each q ∈ todo do
29         ⟨_, k, i⟩ ← T_range.range(q.x, q.y)
30         if k ≠ +∞ then
31             q.dp ← k + 1
32             mark q as next_frontier
33         else  mark ⟨i, q⟩ as new_pivot_pair   // not ready yet
34     Pack points marked as next_frontier into frontier*
35     Pack points marked as new_pivot_pair into pivot*
36     T_pivot.multi_insert(pivot*)
37     Update dp values for q ∈ frontier* in T_range
38     return frontier*
```

We first find all objects q such that $\langle x, q\rangle \in T_{pivot}$, which are all objects with pivots in the frontier (Line 27). We attempt to wake up such object q by searching in T_{range} the half-open rectangle with top-right point as q (Line 29), getting triple $\langle \_, k, i\rangle$, where k is maximum dp^* in the query range and i is the x^* value. If $k \neq +\infty$ (Line 30), meaning no unfinished object in q's lower-left area, then q is ready and we set the DP value of q as $1+k$ (Line 31). Otherwise ($n_\infty \neq 0$), there are still unfinished objects in q's lower-left area, and we reset q's pivot as i, which is selected uniformly at random from all unfinished objects in the queried range.

At the end of a round, all newly-generated pivot-object pairs are inserted into T_{pivot} in parallel (Line 35–36). All newly finished objects are packed into next frontier (Line 34). The DP values of the newly finished objects are updated in T_{range} (Line 37). We use the following lemma to prove the work of the algorithm and prove it in the full version of this paper [62].

Lemma 5.4. *Construct a sequence x_i as follows. Let $x_0 = 1$, and x_i be a uniformly random number selected from x_{i-1} to n. Let k be the first element s.t. $x_k = n$. Then $k = O(\log n)$* whp.

Lemma 5.5. *For each object x in the input sequence of size n, Algorithm 3 will attempt to wake up x for $O(\log n)$ times* whp.

Proof. For an object x, let S be the sequence of objects before x and smaller than x sorted by rank. Let the first pivot of x be the p-th object in S, which is selected uniformly at random from S. When S_p finishes, all objects in S with rank smaller than S_p must have finished, and the next pivot is selected uniformly at random from $S_{p'..n}$, where $p' > p$. Similar process applies to later pivot selections. Therefore, the number of pivots selected for x is no more than the length of sequence in Lemma 5.4, which is $O(\log n)$ *whp*. □

Theorem 5.6. *Algorithm 3 computes the LIS of a sequence of size n in $O(n \log^3 n)$ work and $O(r \log^2 n)$ span* whp*, where r is the rank (LIS length) of the input sequence.*

Proof. First, all initialization including constructing the range tree has work $O(n \log n)$ and span $O(\log^2 n)$. Assume in round i there are n_i objects in the *todo* list, which are the objects that we attempt to wake up. Line 27 finds at most n_i objects from T_{pivot} with $O(n_i \log^2 n)$ work. Each range query costs $O(\log^2 n)$, and thus the total cost in the parallel for-loop in Line 28 is $O(n_i \log^2 n)$. Line 34 packs at most n_i objects with work $O(n_i)$. Line 35 and 36 pack and insert at most n_i elements to T_{pivot}, and thus costs $O(n_i \log^2 n)$. Line 37 updates at most n_i objects to T_{range}, and thus costs $O(n_i \log^2 n)$. In summary, each round of function *wakeup* has work $O(n_i \log^2 n)$. From Lemma 5.5, we know that $\sum n_i = O(n \log n)$. This proves that the total work of Algorithm 3 is $O(n \log^3 n)$ *whp*.

By definition of rank, the number of rounds in Algorithm 3 is r. In each round, the span bounds of range queries (Line 29), pack (Lines 34–35), and update T_{pivot} and T_{range} (Lines 36 and 37) are all $O(\log^2 n)$. Therefore, the total span is $O(r \log^2 n)$. □

5.3 Greedy MIS and Related Applications

In this section, we propose a new parallel MIS (maximal independent set) algorithm. Parallel MIS is widely-studied [3, 13, 23, 24, 28, 29, 32, 33, 41, 42, 56]. Given a graph $G = (V, E)$, an independent set $A \subseteq V$ is a subset of vertices where $\forall u, v \in A$, $(u, v) \notin E$. An MIS is an independent set A where $\forall v \in V, v \notin A$, $A \cup \{v\}$ is not an independent set. The widely-adopted greedy MIS algorithm [13, 23, 28, 29, 42, 56] starts with assigning each vertex a random priority and greedily selecting the vertices based on the priority (highest to lowest). The MIS is initialized as an empty set. When processing vertex v, we add v to the MIS if none of v's neighbor is selected in the MIS, and skip v otherwise. We say a vertex is ***available*** if none of its neighbors are selected in the current MIS, and ***unavailable*** otherwise. Blelloch et al. parallelized the algorithm [13]. Note that a vertex is ready when it has a higher priority than all its available neighbors. We say a neighbor of v is a ***blocking neighbor*** if it has a higher priority than v. When all v's blocking neighbors become unavailable, v is ready. In each round, the parallel algorithm processes all ready vertices in parallel by adding them to the MIS, and removing their neighbors (marking as unavailable). An illustration is shown in Fig. 4(a).

The main challenge in the algorithm is to find ready vertices efficiently. A previous approach [13] hooks each unfinished vertex v to an unfinished neighbor with the highest priority as the pivot. Only when v's pivot is finished, it applies a dual binary search, which either finds the next pivot or decides that v is ready. Combining the results in [13] and [42], the work is $O(m)$ and the span is $O(\log^3 n)$ *whp*. The high span comes from the $O(\log^2 n)$ dual binary search that can apply in each round ($O(\log n)$ steps each requiring an $O(\log n)$-span min-reduce). There exist other algorithms such as Prism [51] which is work-efficient with $O(\log^2 n)$ span. However, it assumes to know the number of processors in the algorithm, and a strong constant-time atomic operation `fetch-and-decrease`.

In this paper, we show a new, asynchronous approach that improves span, while maintaining work-efficiency for parallel MIS. The key idea is to wake up a vertex only when the last blocking neighbor is finished. We use the atomic operation `test_and_set` (TAS), and build a complete binary tree, called the TAS tree, for each vertex, and use it to check if all the blocking neighbors are finished. Let the TAS tree of v be T_v. Each leaf in T_v corresponds to a blocking neighbor u of v (i.e., with a higher priority than v). We set a flag (initialized to *zero*) in each leaf to show if u has been unavailable. Each internal node in the TAS tree also maintains a flag: it is *one* when at least one subtree is fully unavailable, and *zero* otherwise. For example, in Fig. 4(b), ⑭ maintains four blocking neighbors in its TAS tree. Note that when all leaves in the T_v are *one* (unavailable), v is ready to be added to the MIS. We want to use the flag to reflect the unavailability of the subtree. More precisely, for each subtree t, when the last flag in t becomes *one*, the information should be carried to t's parent.

The pseudocode of our MIS algorithm is given in Algorithm 4. The algorithm starts with constructing the TAS trees (initialized to *zero* for all tree nodes), and finding all vertices with an empty TAS tree (the ready ones). We start with processing these vertices in parallel. When processing a vertex, we will mark each of its neighbors u as unavailable. We further need to notify all the TAS trees containing u that u is now unavailable. For each of the TAS trees, we set u's flag in the leaf to be *one* (Line 12). This information is prorogated up along the path to set the flag of its parent, call it p, to be *one* by TAS (Line 17). If the TAS succeeds, it means that the other branch of p's subtree is not fully finished yet, and we can just quit. For example, in Fig. 4, after we process ② in round 1, we set ⑦ and ⑬ as unavailable in ⑭'s TAS tree. Both of them will TAS their parent and succeed, so they quit. When the TAS fails, it means that p's flag is already *one*, so the other branch has been fully unavailable. Since the current subtree at p is also fully unavailable, we will continue to attempt mark p's parent using TAS recursively. For example, in Fig. 4, when we mark ⑫ as unavailable in ⑭'s TAS tree, we first TAS its parent. As it was set by ⑬ previously, the TAS fails. We then continue to its parent (the root). This TAS succeeds.

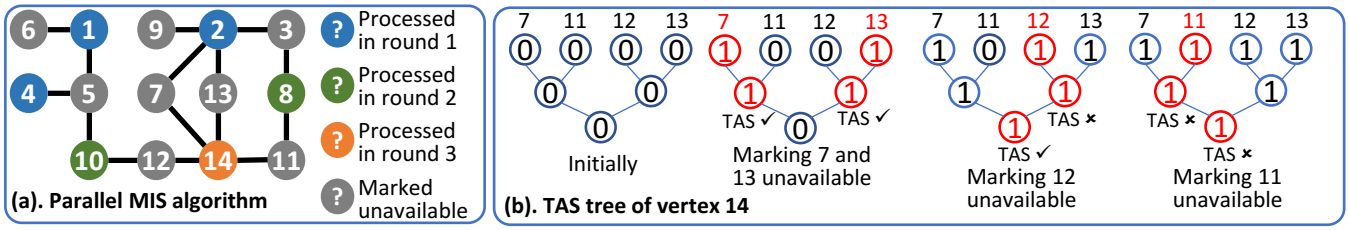


Figure 4: Illustration of the greedy maximal independent set (MIS) algorithm. (a). The input graph (numbers are priorities) and the round in which each vertex is processed in the algorithm. (b). The TAS tree of ⑭ in the graph in (a) from the initial status to when marking some leaves unavailable. ✓= successful TAS. ✗ = unsuccessful TAS. When there is an unsuccessful TAS at the root, the entire TAS tree is finished.

Algorithm 4: The parallel MIS algorithm.

Input: A graph $G = (V, E)$ with priority $p : V \mapsto \mathbb{Z}$.
Output: Maximal Independent Set of G

```
1  parallel_for_each v ∈ V do
2      T_v ← the TAS tree for v
3      Maintain the list of TAS trees that contains each vertex
4      status[v] ← undecided
5  parallel_for_each v ∈ V do
6      if T_v is empty then WakeUp(v)  // no blocking neighbor
7  return all v ∈ V marked as selected
8  Func WakeUp(v)
9      status[v] ← selected
10     parallel_for_each u ∈ N(v) do
11         status[u] ← removed
12         parallel_for_each TAS tree T_v contains u do
13             if status[v] ≠ removed then
14                 x ← the leaf of u in T_v
15                 x.flag ← true
16                 p ← parent(x)
17                 while test_and_set(p.flag) successful do
18                     if p is the root of T_u then
19                         WakeUp(v)
20                         Break
21                     p ← parent(p)
```

When a TAS at the root of any TAS tree T_v fails, the entire subtree is now unavailable, so v is ready to be waked up (Line 19), and we will repeat this process for v. For example, when we mark ⑪ as unavailable, we TAS its parent and fail, and continue to the parent, which is the root. This TAS also fails, which means the entire tree is unavailable. Therefore, ⑭ can be waked up.

Theorem 5.7. *Algorithm 4 generates the greedy maximal independent set of $G = (V, E)$ in $O(m)$ work and $O(\log n \log d_{max})$ span* whp, *where $n = |V|$, $m = |E|$, and d_{max} is the maximum degree in G.*

We note that our new MIS algorithm is fully asynchronous: no round-based synchronization is used. Although this does not directly follow our phase-parallel algorithm, it also uses the same idea of our Type 2 framework, where we wish to identify the last finished object in $\mathcal{P}(x)$ and wake up x at that time. The rank of each vertex v can be viewed as the longest chain with decreasing priority starting from v. We now analyze the cost of this algorithm.

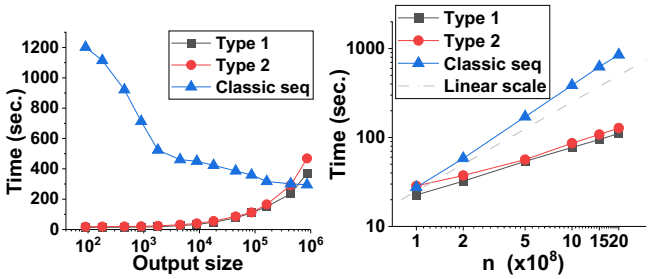
We show the proof of Thm. 5.7 in the full version of this paper [62]. In the worst case when $d_{max} = n$, the span of Algorithm 4 is $O(\log^2 n)$, which improves previous result by a factor of $O(\log n)$. Comparing to Prism [51], our approach requires no knowledge on the number of processors and is completely in the fork-join model. The efficiency of our algorithm comes from the simple idea TAS tree data structure. We note that using tree-like structure to do counting is used in previous work [40, 44], but our algorithm is different in that we make the observation that in the MIS algorithm, the information needed is *whether* all blocking neighbors all finish, instead of the number of unfinished blocking neighbors. This simplifies the problem and enables better span bound.

Graph Coloring and Matching. Several iterative graph algorithms that share the similar approach can be improved using the same technique. For graph coloring, Jones and Plassmann [49] showed the greedy algorithm that can be parallelized using the similar approach in [13]. Hasenplaugh et al. [47] analyzed a list of heuristics that vary the greedy order of the vertices and can lead to different span bounds and output quality. For graph matching, Blelloch et al. [13] showed a parallel greedy algorithm that is very similar to the MIS algorithm in the same paper. By replacing the original wake-up strategy in [13] with our new approach, we can improve the span by $O(\log n)$. We note that the parallel graph-matching algorithm cannot be fully asynchronous since each edge's readiness relies on two vertices, which needs to be checked after synchronization. Hence, a synchronization is required between the rounds, but that does not change the span bound. The analysis by Hasenplaugh et al. [47] assumes atomic decrement-and-fetch operations which is usually not included on the family of the binary-forking models [12]. Using the technique in this paper can achieve the same bounds without this assumption.

Other Algorithms. Many algorithms in [12, 16–18] have constant size $\mathcal{P}(x)$ for all $x \in S$. Blelloch et al. [12] showed that `test_and_set` can be used to check the readiness in this case. Here we note that this can be considered as a special case for our TAS trees, just with constant sizes. These applications include random permutation, list ranking, and tree contraction [12, 64]; convex hull and Delaunay triangulation [16–18]. Although we do not improve the bounds of these algorithms, we show the interconnections among all these algorithms, and an additional angle to review these problems and algorithms.

6 EXPERIMENTS

In addition to the new theoretical results, we also implemented many proposed algorithms based on our phase-parallel frameworks, including activity selection (both Type 1 and Type 2), Huffman tree, SSSP, and LIS. We use our experiments as proofs-of-concept to show how work-efficiency and round-efficiency affect performance in practice. We run our experiments on a 96-core (192 hyperthreads) machine with four Intel Xeon Gold 6252 CPUs, and 1.5 TiB of main memory. Our implementation is in C++ with Cilk Plus [55]. For the parallel results, we use all cores and fully interleave the memory



(a). $n = 10^9$, varying rank (b). $r = 45000$, varying input size

Figure 5: Experiments on activity selection. (a). fix input size $n = 10^9$ and vary the rank of the input. (b). fix rank $r = 45000$ and vary the input size. "Classic seq" is the classic sequential DP algorithm. "Linear scale" shows the slope of linearly growing line.

among NUMA areas using numactl -i all. We use r as the input rank. All reported numbers are the averages of the last five runs among six repeated experiments. Due to page limitation, we put some extra experimental results and discussions about LIS and Huffman tree in the full version of this paper [62].

6.1 Activity Selection

We implement Type 1 and Type 2 algorithms for activity selection. We also implement a sequential version based on the DP recurrence in Eq. (1) for comparison. For each activity, we set a random start time and a length based on a truncated normal distribution. We control the mean and standard deviation of this distribution to control the rank of the input data. The weights are generated uniformly at random in $[1, 2^{32})$.

Fig. 5(a) shows the running time of all tested algorithms on 10^9 input activities with various ranks. Both our Type 1 and Type 2 algorithms have very similar performance and outperform the classic sequential algorithm up to rank of 4×10^6. Note that the running time of our algorithm increases as the rank increases because our algorithms have span proportional to the rank. Also, when the rank is large, each round in the algorithm only deals with a small number of objects, which also harms parallelism. Even so, the running time of our algorithm seems to grow sublinearly with the rank. This is because with about 200 threads, the work should still dominate the cost, and both our algorithms are work-efficient. Interestingly, the performance of the classic sequential algorithm improves with increasing input rank. This is because in the sequential algorithm, when the rank is large, the range query of an activity x (see Eq. (1)) will likely find an activity close to x, which exhibits better cache locality. For small ranks, our algorithms can be up to 80x faster than the sequential algorithm. For rank of $O(\sqrt{n})$, our algorithm is still about 14x than the classic sequential algorithm.

Fig. 5(b) shows the running time of all tested algorithms on different input sizes with a fixed rank of $r = 45000$. For other values of the rank, we see similar trends, so we just show $r = 45000$ as an example. Our algorithm scales well to large input sizes. The sequential algorithm grows superlinearly with the input size n, which matches the theoretical cost $O(n \log n)$. As shown in Fig. 5(b), our algorithm grows much slower with n (almost linearly). This is because when n increases, the number of activities to be processed within each round also increases, which overall enables better parallelism. We believe that this indicates the potential of our algorithm to scale to even larger data and more threads.

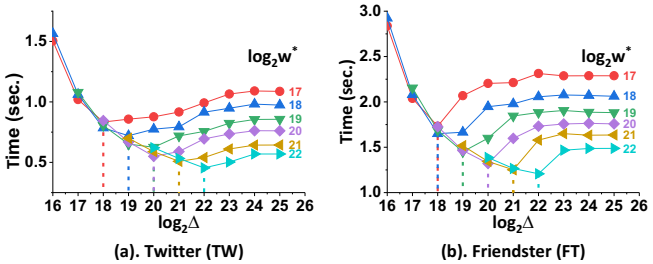


Figure 6: Experiments on parallel SSSP. (a). Running time on graph Twitter (41.7 millions of vertices and 1.47 billions of directed edges). (b). Running time on graph Friendster (65.6 millions of vertices and 3.61 billions of undirected edges). We use Δ-stepping implementation from [39] with different values of Δ and w^* (the smallest edge weight in the graph). Values on the right are the values of $\log_2 w^*$. The maximum edge weight is 2^{23}.

On all tests, Type 1 algorithm outperforms the Type 2 algorithm by up to 35%. This is because in the Type 2 algorithm, we need to first find pivots for all activities, while in the Type 1 algorithm, we directly start with processing all ready activities, and use a range query to find the frontier. Nevertheless, the two algorithms still have very similar performance and both outperform the classic sequential algorithm for reasonably large input rank.

6.2 Parallel SSSP

Background. SSSP is a challenging problem in the parallel setting. As mentioned, Dijkstra is work-efficient but hard to parallelize. Bellman-Ford has better parallelism but significantly more work. Even so, parallel Bellman-Ford has good performance on low-diameter graphs such as social networks [39, 63] due to better parallelism. In practice, many heuristics were proposed aiming to achieve a tradeoff between work and parallelism. For example, the Δ-stepping algorithm [57] determines the correct shortest distances of vertices in increments of Δ of the tentative distances. In step i, the algorithm will find and settle down all the vertices with distances in $[i\Delta, (i+1)\Delta]$. Δ-stepping is highly practical and widely used but its performance is very sensitive to the parameter Δ [39].

Our experiments. As mentioned in Sec. 4.2, our phase-parallel algorithm settles down all vertices within tentative distance $[iw^*, (i+1)w^*]$, where w^* is the smallest edge weight. This is conceptually the same as using $\Delta = w^*$ in Δ-stepping [57]. Therefore, we run experiments using the Δ-stepping implementation by Dong et al. [39] to test our idea. We note that this is not exactly the same as our algorithm as the implementation does not use a tree-based data structure to extract the frontier[3], but is still "work-efficient" w.r.t. the number of total relaxations. We note that empirically, the I/O cost in processing and relaxing edges is usually the main cost in practical parallel SSSP algorithms. Our results highly match our theory. We tested two graph benchmarks, Twitter [54] and Friendster [69]. Both of them are real-world large-scale social networks with small diameters.

In our experiments, we fix the largest edge weight as $w_{max} = 2^{23}$, vary the w^* from 2^{17} to 2^{22}, and set the weight uniformly at random in this range for each edge[4]. For each edge weight range,

[3]In fact, almost none of the parallel SSSP implementation uses tree-based structures to maintain distances due to their worse cache locality than flat arrays.
[4]This setting is similar to *weighted BFS* [35] (which is trivially work-efficient). The difference is that after normalizing, the edge weight in our problem are not necessarily integers as in weighted BFS.

we run Δ-stepping with Δ varying from 2^{16} to 2^{26}. We show the running time in Fig. 6. For both graphs, the best choice of Δ almost exactly matches w^* (differ by at most 2x), when w^* is close to w_{max}. This verifies the importance of work-efficiency in practical SSSP algorithms. When w^* gets even smaller, using $\Delta = w^*$ does not perform well, which is also as expected—despite work-efficiency, using a small Δ limits the frontier size, hence we cannot fully exploit parallelism. This also reveals the work-parallelism (or work-round) tradeoff. In all, when w_{max}/w^* is within 32, using our algorithm (i.e., $\Delta = w^*$) gives reasonably good performance.

It is worth noting that we also tried the same algorithm on large-diameter graphs, such as some road graphs [1]. Probably not surprisingly, even on $w^* = w_{max}/2$, $\Delta = w^*$ did not give the best performance. This is because on such graphs, the frontier size is usually small, and the performance is usually limited by the lack of parallelism. In this case, avoiding extra work does not help much in improving performance (and even harms the performance since the parallelism gets worse). Many state-of-the-art implementations [39, 70] optimize performance on such (large-diameter) graphs by sacrificing more work to get better parallelism.

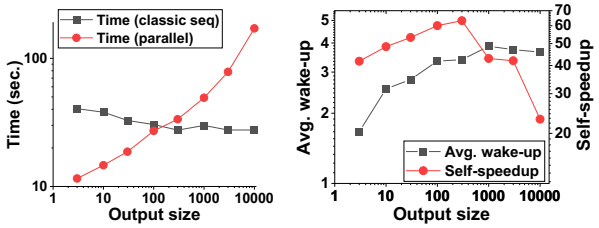
6.3 Parallel LIS

When implementing our LIS algorithm, we use nested arrays to represent augmented range trees to improve locality. We also use a heuristic when choosing pivots—instead of choosing a uniformly random pivot, we choose the right-most unfinished point as the pivot, according to the intuition that points to the right are more likely to be processed in later rounds. We also implemented a standard $O(n \log n)$ sequential version based on Eq. (2).

We test input data of 10^8 with different ranks (LIS sizes). We present results for LIS implementations in Fig. 7 and Fig. 8. We also vary the input rank, which is the LIS size. We use two different data patterns. The first one is roughly k segments of data, and we call it the *segment pattern*. The data values are roughly decreasing with random noise within each segment and roughly increasing across the segments. The LIS size is about k. The other pattern is generated by an increasing line $a_i = t \times i + b_i$, where b_i is a random variable chosen from a uniform distribution. We call it the *line pattern*. By changing the slope t and the distribution of b, we can also control the rank of the input data. We visualize the input data pattern in the full version of this paper [62].

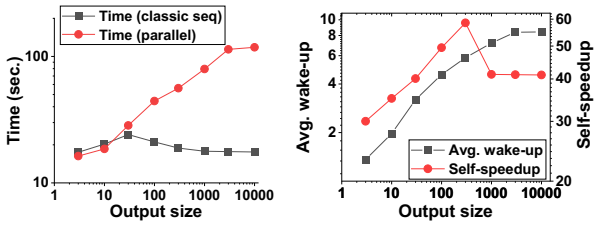
We show our running time as well as the average number of wake-up attempts for all objects. On the two different data patterns, our algorithm is faster before the rank r is up to about 100, and perform worse than the sequential algorithm afterward. Note that our algorithm has an $O(\log^2 n)$ overhead in work. When the parallelism is not sufficient to compensate for the overhead in work, the performance may drop. Interestingly, though our algorithm is getting slower as the rank increases which matches theory, the standard sequential algorithm is getting faster. We believe the improvement in the sequential algorithm is also caused by better locality because the range query for an object x will find an object close to x.

We note that our algorithm still shows very good scalability—in most tests, our self-speedup is more than 40x. The poor performance with the large rank comes from the work-inefficiency. Even the polylogarithmic overhead ($\log^2 n$) can be more than the number



(a). Time (b). Average wake-up and self-speedup

Figure 7: Experiments on LIS (segment). We fix input size $n = 10^8$ and use the *segment pattern*, with varying the output size. "Classic seq" is the classic sequential DP algorithm.



(a). Time (b). Average wake-up and self-speedup

Figure 8: Experiments on LIS (line). We fix input size $n = 10^8$ and use the *line pattern*, with varying the output size. "Classic seq" is the classic sequential DP algorithm.

of available processors on our machine. Indeed, our algorithm's sequential running time (on one core) is much more than the standard sequential algorithm. Therefore, when k is large, the parallelism cannot compensate the overhead due to work-inefficiency.

We also observed that the average number of wake-ups is very small. In all our tests, the maximum value is 8 times, which is less than $\log n$ shown in Lem. 5.5. This is partially enabled by our heuristic. Especially for the segment pattern, when choosing the right-most unfinished object as the pivot, it is almost always the last blocking object to wait. We believe our algorithm is scalable to more cores, but we are also interested in improving the work bound to closer to work-efficient. Reducing work-bound should be promising to improve practical performance.

7 CONCLUSION AND FUTURE WORK

In this paper, we used the phase-parallel framework with general techniques to parallelize sequential iterative algorithms with certain dependences, and designed work-efficient and round-efficient algorithms for a variety of classic problems. Our results improved the previous theoretical bounds for many of them (e.g., the LIS and MIS algorithms). We also implemented these algorithms. Our results illustrated how work-efficiency and round-efficiency affect performance in practice (which matches our theory). For reasonable (not too large) input ranks, our work-efficient algorithms achieved good parallelism and outperformed the sequential algorithms. One interesting future direction is to reduce the $O(\log^2 n)$ overhead in the work of the LIS algorithm, which is also likely to improve its practical performance. We leave this as future work.

ACKNOWLEDGEMENT

This work is supported by NSF grant CCF-2103483.

REFERENCES

[1] Openstreetmap © openstreetmap contributors. https://www.openstreetmap.org/, 2010.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theoretical Computer Science (TCS)*, 35(3), 2002.

[3] Y. Afek, N. Alon, Z. Bar-Joseph, A. Cornejo, B. Haeupler, and F. Kuhn. Beeping a maximal independent set. *Distributed computing*, 26(4):195–208, 2013.

[4] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[5] M. R. Alam and M. S. Rahman. A divide and conquer approach and a work-optimal parallel algorithm for the lis problem. *Information Processing Letters*, 113(13):470–476, 2013.

[6] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[7] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[8] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008.

[9] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[10] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2012.

[11] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[12] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[13] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[14] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2004.

[15] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[16] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[17] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. *J. ACM*, 2020.

[18] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[19] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1998.

[20] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. *Theory of Computing Systems (TOCS)*, 32(3), 1999.

[21] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[22] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[23] N. Calkin and A. Frieze. Probabilistic analysis of a parallel algorithm for finding maximal independent sets. *Random Structures & Algorithms*, 1(1):39–50, 1990.

[24] S. Chatterjee, R. Gmyr, and G. Pandurangan. Sleeping is efficient: Mis in o (1)-rounds node-averaged awake complexity. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 99–108, 2020.

[25] R. Chowdhury, P. Ganapathi, Y. Tang, and J. J. Tithi. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 339–350, 2017.

[26] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911–925, 2013.

[27] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing (TOPC)*, 3(4), 2017.

[28] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Inf. Control*, 64, March 1985.

[29] D. Coppersmith, P. Raghavan, and M. Tompa. Parallel graph algorithms that are efficient on average. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 260–269. IEEE, 1987.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[31] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, pages 722–731. Springer, 1998.

[32] J. Dahlum, S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. Accelerating local search for the maximum independent set problem. In *International symposium on experimental algorithms*, pages 118–133. Springer, 2016.

[33] S. Daum, M. Ghaffari, S. Gilbert, F. Kuhn, and C. Newport. Maximal independent sets in multichannel radio networks. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 335–344, 2013.

[34] L. Dhulipala, G. E. Blelloch, Y. Gu, and Y. Sun. Pac-trees: Supporting parallel and compressed purely-functional collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[35] L. Dhulipala, G. E. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[36] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun. Semi-asymmetric parallel graph algorithms for nvrams. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9), 2020.

[37] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1), 1959.

[38] D. Dinh, H. V. Simhadri, and Y. Tang. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[39] X. Dong, Y. Gu, Y. Sun, and Y. Zhang. Efficient stepping algorithms and implementations for parallel shortest paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2021.

[40] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.

[41] J. T. Fineman, C. Newport, M. Sherr, and T. Wang. Fair maximal independent sets. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 712–721. IEEE, 2014.

[42] M. Fischer and A. Noever. Tight analysis of parallel randomized greedy mis. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2152–2160, 2018.

[43] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. *J. Parallel Distrib. Comput.*, 21(2), 1994.

[44] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 64–75, 1989.

[45] P. Gruevski, W. Hasenplaugh, D. Lugato, and J. J. Thomas. Laika: Efficient in-place scheduling for 3d mesh graph computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 415–426, 2018.

[46] Y. Gu, O. Obeya, and J. Shun. Parallel in-place algorithms: Theory and practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 114–128, 2021.

[47] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[48] S. Im, B. Moseley, and X. Sun. Efficient massively parallel methods for dynamic programming. In *ACM Symposium on Theory of Computing (STOC)*, pages 798–811, 2017.

[49] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. 14(3):654–669, 1993.

[50] M. Kainer and J. L. Träff. More parallelism in dijkstra's single-source shortest path algorithm. *arXiv preprint arXiv:1903.12085*, 2019.

[51] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. *ACM Transactions on Parallel Computing (TOPC)*, 3(1):1–31, 2016.

[52] P. Krusche and A. Tiskin. Parallel longest increasing subsequences in scalable time and memory. In *International Conference on Parallel Processing and Applied Mathematics*, pages 176–185. Springer, 2009.

[53] P. Krusche and A. Tiskin. New algorithms for efficient parallel string comparison. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 209–216, 2010.

[54] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[55] C. E. Leiserson. Cilk. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011.

[56] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. on Computing*, 15, 1986.

[57] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[58] T. Nakashima and A. Fujiwara. Parallel algorithms for patience sorting and longest increasing subsequence. In *International Conference in Networks, Parallel and Distributed Processing and Applications*, pages 7–12, 2002.

[59] T. Nakashima and A. Fujiwara. A cost optimal parallel algorithm for patience sorting. *Parallel processing letters*, 16(01):39–51, 2006.
[60] X. Pan, D. Papailiopoulos, S. Oymak, B. Recht, K. Ramchandran, and M. I. Jordan. Parallel correlation clustering on big graphs. In *Advances in Neural Information Processing Systems (NIPS)*, pages 82–90, 2015.
[61] D. Semé. A cgm algorithm solving the longest increasing subsequence problem. In *International Conference on Computational Science and Its Applications*, pages 10–21. Springer, 2006.
[62] Z. Shen, Z. Wan, Y. Gu, and Y. Sun. Many sequential iterative algorithms can be parallel and (nearly) work-efficient. *arXiv preprint arXiv:2205.13077*, 2022.
[63] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2013.
[64] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 431–448, 2015.
[65] Y. Sun and G. E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019.
[66] Y. Sun, D. Ferizovic, and G. E. Blelloch. Pam: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.
[67] G. Thierry, M. Jean-Frédéric, and S. David. A work-optimal cgm algorithm for the lis problem. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 330–331, 2001.
[68] A. Tiskin. Fast distance multiplication of unit-monge matrices. *Algorithmica*, 71(4):859–888, 2015.
[69] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.
[70] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.